PATENT APPLICATION
DOCKET NO. RIDG101

COMPRESSED DATA STRUCTURE AND DECOMPRESSION SYSTEM

INVENTORS:

Patrick M. Sewall
Russell Campbell
Dan George
Lon Radin

COMPRESSED DATA STRUCTURE AND DECOMPRESSION SYSTEM

FIELD OF THE INVENTION

[0001]   The present invention is directed to a compressed data structure and a method and system for decompressing the data structure.

BACKGROUND OF THE INVENTION

[0002]   Throughout the years, the architecture of computing systems has remained substantially unchanged. Typically, at the heart of a computing system is a central processing unit (CPU) coupled to a system memory. The system memory is usually volatile and is erased when the system is powered down. The computing system also includes a separate graphical interface for coupling to a video display as well as input/output (I/O) control logic for various I/O devices, including a keyboard, mouse, floppy drive, hard drive, among others.

[0003]   Fig. 1 illustrates a typical computing system 10 including CPU 12, volatile memory 14, and non-volatile memory 16. Volatile memory 14 represents generally any memory, such as DRAM (Dynamic Random Access Memory) and SRAM (Static Random Access Memory), that loses its contents when the power is turned off. Non-volatile memory represents generally an I/O device such as a floppy disk, hard drive, or other form of memory that is retained even when power is turned off. In general, the operation of computing system 10 is as follows. Stored programs 18 and data 20 are read from non-volatile memory 16 and transferred into volatile memory 14 and are shown as programs and data in use 22 and 24. CPU 12 then executes program in use 22 from volatile memory 14. CPU 12 processes the instructions of program 22, and the resulting data is provided to one or more of the various output devices (not shown). Computing system 10 may include several output devices, including a video display, audio speakers, and printers, among many others.

[0004]   In the past, such computing systems were usually only found in personal computers, servers, and other conventional computing devices. Today, however, computing systems are being embedded in appliances such as microwave ovens and refrigerators. They are also being embedded in portable devices such as cellular telephones, digital music players (commonly referred to as MP3 players), and PDA's

(Personal Digital Assistants). It is with these portable devices that power consumption and available memory pose significant constraints on the computing system.

[0005]   Most portable devices operate on battery power. Replacing or recharging batteries is a disagreeable task for any user. Consequently, extended battery life is an important, if not essential, marketing tool. Conventional nonvolatile memory sources, such as hard drives, are not practical for many portable devices. Hence, flash memory is commonly used for nonvolatile memory in portable devices. Flash memory is a special type of EEPROM (Electrically Erasable Programmable Read-Only Memory) that can be erased and reprogrammed in blocks instead of one byte at a time. Flash memory, like a hard drive, retains its contents even when power is turned off. Presently, flash memory is a relatively expensive option. Computing systems requiring large amounts of flash memory can dramatically increase the manufacturing cost of a portable device.

[0006]   To decrease required flash memory, programs and other electronic data files are stored in a compressed state that much less space than the decompressed counterparts. When these programs and other data files need to be loaded for use, they are decompressed "on the fly" as part of the loading process. Consequently, what is needed is a compressed data structure and decompression system that reduces power consumption and minimizes the amount of required non-volatile memory.

SUMMARY OF THE INVENTION

[0007]   The present invention is directed to a compressed data structure and decompression system suitable for use in portable devices. The data structure and decompression system of the present invention help reduce power consumption and minimize the amount of non-volatile memory needed to support a computing system. In one embodiment of the invention, the data structure contains a plurality of code strings and a plurality of look-up strings. Each look-up string includes an index identifying a particular code string to be retrieved and an instruction identifying an operation to be performed on the retrieved code string. To decompress the data structure, software or other programming reads through each of the look-up strings. For each look-up string the programming retrieves a code

string and performs an operation on that code string according to the index and instruction of the look-up string.

DESCRIPTION OF THE DRAWINGS

[0008]   Fig. 1 is a schematic representation of a typical computing system.

[0009]   Fig. 2 is a schematic representation of a computing system in which the present invention may be embodied.

[0010]   Fig. 3 is a schematic representation of a library according to one embodiment of the present invention.

[0011]   Fig. 4 is a schematic representation of a library segment in the library of Fig. 3.

[0012]   Fig. 5 is a schematic representation of a code string according to one embodiment of the present invention.

[0013]   Fig. 6 is a schematic representation of a history cache according to one embodiment of the present invention.

[0014]   Fig. 7 is a schematic representation of look-up data according to one embodiment of the present invention.

[0015]   Fig. 8 is a schematic representation of a look-up string in the look-up data of Fig. 7.

[0016]   Fig. 9 is a schematic representation of a decompression engine according to one embodiment of the present invention.

[0017]   Fig. 10 is a flow diagram illustrating a data decompression process provided by the decompression engine of Fig. 9.

[0018]   Fig. 11 is a block diagram illustrating an example of a library designed for an ARM (Advanced RISC Machine) processor.

[0019]   Fig. 12 is a block diagram further illustrating the first library segment of the library of Fig. 11.

[0020]   Fig. 13 is a table illustrating look-up strings designed to retrieve a code string from the library of Fig. 11 and write it to output memory.

[0021]   Fig. 14 is a table illustrating look-up strings designed to retrieve a code string from the library, alter that code string, and then and write the altered code string to output memory.

[0022]   Fig. 15A is a table illustrating look-up strings designed to retrieve a code string from the history cache and write it to output memory.

[0023]   Fig 15B is a second table illustrating look-up strings designed to retrieve a code string from the library and write it to output memory.

[0024]   Fig. 16 is a table illustrating look-up string designed to write the index of the look-up string to output memory.

DETAILED DESCRIPTION OF THE INVENTION

[0025]   *INTRODUCTION:* The idea behind data compression is relatively simple. Efficiently compressing and then decompressing a data file or files, however, is another story. Every electronic data file, such as an application, electronic document, or other electronic information, consists of a binary string of bits – ones and zeros. A given string of bits can be broken into a series of code strings in the form of words, bytes, or other grouping of bits. A word is a string of thirty-two bits and a byte is a string of eight bits. Using words as an example, within a string that forms a given data file, some words are repeated, and some of those words are repeated more then others. Compression involves representing each of these repeated words with a representation string containing, in this example, fewer than thirty-two bits. If, for a given program, every word (thirty-two bits) could be represented with a byte (eight bits), then that program could potentially be compressed to one quarter its original size. Generally, however, representing each of the repeated words in a given data file requires representation strings of varying length. Representation strings containing the fewest bits are used to represent the most frequently repeated words while representation strings containing more bits are used to represent the least frequently repeated words.

[0026]   A compressed data structure, according to the present invention, includes look-up data and a library. The library contains a number of entries – each entry being a repeated code string. The look-up data contains look-up strings.   Each look-up string functions in part as a representation string that identifies an entry in the library or elsewhere. Each look-up string also includes data identifying an operation to be performed on the library entry identified by the look-up string. Each code string needed to make up the decompressed data file or files can be identified by reading through each of the look-up strings. Joining each of the identified code strings, then, forms the decompressed data file or files.

[0027]   *COMPONENTS*: Although the various embodiments of the invention disclosed herein will be described with reference to the computing system 26

shown schematically in Fig. 2, the invention is not limited to use with computing system 26. The invention may be implemented in or used with any system in which it is necessary or desirable to provide and decompress data structures. The following description and the drawings illustrate only a few exemplary embodiments of the invention. Other embodiments, forms, and details may be made without departing from the spirit and scope of the invention, which is expressed in the claims that follow this description.

[0028]   The logical components of one embodiment of the compressed data structure and decompression system will now be described with reference to the block diagrams of Figs. 2-9. Fig. 2 illustrates computing system 26 containing CPU 28, volatile memory 30, non-volatile memory 32, and system bus 34. System bus 34 represents a communication link between CPU 28 and the other components 30 and 32. In this embodiment, the invention is implemented in compressed data structure 36 stored in non-volatile memory 32 as well as software or other programming labeled decompression program 38 operating from volatile memory 30. Compressed data structure 36 represents generally a software application or other electronic data stored in a compressed format. Decompression program 38 represents software or other programming when executed by CPU 28 is capable of decompressing compressed data structure 36. The term decompression engine, then refers generally to decompression program 38 being executed by CPU 28. One embodiment of decompression program 38 will be described in more detail below with reference to Fig. 9.

[0029]   Still referring to Fig. 2, volatile memory 30 also contains history cache 40 and output memory 42. Output memory 42 represents a portion or portions of volatile memory 30 into which the program or other electronic data represented by compressed data structure 36 is decompressed by CPU 28 using decompression program 38. CPU 28, using decompression program 38, sequentially retrieves code strings from compressed data structure 36 and places those code strings into output memory 42. The retrieved code strings are joined to form the original/decompressed decompressed data file represented by compressed data structure 36. History cache 40 represents a memory location capable of containing copies of a specified number of the most recently retrieved code strings. The number of copies is typically determined by the amount of memory allocated to or

available for history cache 40. History cache 40 will be described in more detail below with reference to Fig. 6.

[0030] As illustrated in Fig. 2, CPU 28 includes processor 44 and processor cache 46. Processor 44 is responsible for executing decompression program 38 as well as other programs contained in volatile memory 30 and elsewhere as well as accessing data from volatile and non-volatile memory 30 and 32. In general, a program is made up of a series of instructions each guiding CPU 28 to perform a particular task. As a program is being executed, particular instructions and data are often times repeatedly processed by CPU 28. Processor cache 46 represents memory for containing those instructions and data most often processed within a given time frame. As illustrated in Fig. 2, it is envisioned that processor cache 46 will be L1 (Level One) directly accessed by processor 44. This allows processor 44 to access the cached data at a rate or speed set by CPU 28. Typically the speed of CPU 28 is significantly greater than that provided by system bus 34. Allowing processor 44 to access this data and these instructions directly from processor cache 46, in many cases, significantly increases the overall speed of computing system 26. Increasing the overall speed while decreasing the frequency at which CPU accesses instructions and other data from memory 30 and 32 also decreases power consumption

[0031] In Fig. 2, compressed data structure 36 includes look-up data 48 and library 50. Library 50, described in more detail below with reference to Figs. 3-5, represents a grouping of code strings. As described above, decompression program 38 retrieves these code strings in a particular order joining them in output memory 42. Look-up data 48, described in more detail below with reference to Figs. 7 and 8, represent data used to determine the particular order that the code strings are retrieved from library 50 and history cache 40 as well as instructions for altering a particular one code string before it is joined in output memory 42.

[0032] Referring now to Figs. 3-5, library 50 is divided into multiple segments 52 (Fig. 3). Each segment 50 contains any number of code strings 54 each associated with an indexed position 56 (Fig. 4). Each code string 54 contains a selected number of bits 58 (Fig. 5). A particular code string 54 can be located within library 50 by ascertaining its address – that being its indexed position 56 and segment 52. As shown in Fig. 6, history cache 40 contains a selected number of indexed

positions 60 with each position containing a code string 54 recently written to output memory 42. As noted above, the number of indexed positions 60 depends upon the amount of memory allocated to or available for history cache 40. History cache 40 provides a record of code strings 54 recently written to output memory 42. Copies of at least some of the code strings 54 written to output memory 42 are also written to history cache 40. As history cache 40 is filled to capacity, older code strings 54 are flushed to make room for the new. In the example of Fig. 6, as a new code string 54 is written to output memory 42, the existing code strings 54 in history cache 40 are each shifted down one indexed position 60. The code string 54 in the last indexed position, S in this case, is flushed. The code string 54 written to output memory 42 is then written to the now empty first indexed position 60.

[0033]   Look-up data 48, as shown in Fig. 7, includes a series of look-up strings 64. Look-up strings 64 each represent data used to locate a particular code string 54 as well as data identifying an operation to be performed on that code string 54 by CPU 28 when executing decompression program 38. Described in greater detail below, these operations can include writing the code string 54 one or more times to output memory 42 or first altering code string 54 and then writing the altered code string 54 to output memory 42.

[0034]   Fig 8 illustrates one look-up string 64 in greater detail. Look-up string 64 is divided into two parts – instruction 66 and index 68. Instruction 66 represents data identifying an operation to be performed on a code string 54. If the code string 54 is to be retrieved from library 50 rather than history cache 40, instruction 66 also includes data identifying the library segment 52 from which the code string 54 is to be retrieved. Depending upon the operation to be performed on the retrieved code string 54, index 68 can represent data identifying the indexed position 56 of code string 54 within a particular library segment 52, the indexed position 60 of the code string 54 within history cache 40, as well as data representing a particular operation to be performed on the code string 54. As illustrated each look-up string 64 is in binary format with each instruction 66 and index 68 containing a selected number of bits. The number of bits in each is dependent upon a number of factors including the location of the code string 54 to be retrieved and the particular operation to be performed on that code string 54.

[0035]   Referring now to Fig: 9, decompression program 38 includes look-up string reader 70, code string retriever 72, code string manipulator 74, and code string writer 76. Look-up string reader represents generally any programming enabling CPU 28 to sequentially read look-up strings 64 from look-up data 48. Code string retriever 72 represents any programming enabling CPU 28 to locate and retrieve a code string 54 identified using look-up string reader 70. Code-string manipulator 74 represents any programming enabling CPU 28 to alter a code string 54 as indicated by instruction 66 of look-up string 64. Code string writer 76 represents any programming enabling CPU 28 to write a code string 54 to history cache 40 and to output memory 42. When writing to output memory 42, code string writer 76 appends each code string 54 to code strings previously written to output memory 42. When writing to history cache 40, code string writer 76 shifts existing code strings 54 in history cache 40 down one indexed position 60 and writes the new codes string 54 to the first indexed position 60.

[0036]   The block diagrams of Figs. 2-9 show the architecture, functionality, and operation of one implementation of the present invention. If embodied in software, each block may represent a module, segment, or portion of code that comprises one or more executable instructions to implement the specified logical function(s). If embodied in hardware, each block may represent a circuit or a number of interconnected circuits to implement the specified logical function(s).

[0037]   Also, the present invention can be embodied in any computer-readable medium for use by or in connection with an instruction execution system such as a computer/processor based system or other system that can fetch or obtain the logic from the computer-readable medium and execute the instructions contained therein. A "computer-readable medium" can be any medium that can contain, store, or maintain decompression program 38 and compressed data structure 36 for use by or in connection with the instruction execution system. The computer readable medium can comprise any one of many physical media such as, for example, electronic, magnetic, optical, electromagnetic, infrared, or semiconductor media. More specific examples of a suitable computer-readable medium would include, but are not limited to, a portable magnetic computer diskette such as floppy diskettes or hard drives, a random access memory (RAM), a read-only memory (ROM), an erasable programmable read-only memory, or a portable compact disc.

[0038] *OPERATION:* The operation of decompression program 38 and use of compressed data structure 36 will now be described with reference to the flow diagram of Fig. 10 as well as the preceding Figs. 2-9. Executing decompression program 38, CPU 28 accesses non-volatile memory 32 and retrieves the first look-up string 64 from look-up data 48 (step 80). From that look-up string 64, CPU 28 then identifies a code string 54 to retrieve and an operation to perform on the retrieved code string 54 (step 82).

[0039] Steps 84A through 84C represent actions taken by CPU 28 depending upon the operation identified in the particular look-up string 64. In step 84A, CPU 28 retrieves the code string 54 from the segment 52 and index position 56 of library 50 as identified by the look-up string 64. In step 84A, CPU 28 retrieves and alters the code string 54 from the segment 52 and index position 56 of library 50 as identified by the look-up string 64. In step 84C, CPU 28 retrieves the code string 54 from the index position 68 of history cache 40 as identified by the look-up string 64. In step 84D, CPU 28 retrieves and alters the code string 54 from the index position 68 of history cache 40 as identified by the look-up string 64. Finally, in step 84E, CPU 28 writes the data contained in the index 68 of the look-up string 64 directly to output memory 42 joining that data to data previously written to output memory 42. In step 86, CPU 28 writes the code string 54 retrieved in steps 84A, B, C, or D and altered in steps 84B or 84D to output memory 42, joining that code string to data previously written to output memory 42. CPU 28 then writes the code string 54 to history cache 40 (step 88).

[0040] If the current look-up string 64 is the last string in look-up data 48, CPU 28 executes the program formed by each of the code strings written and joined in output memory 42 (step 90). Otherwise, CPU 28 repeats the process with step 80 retrieving the next look-up string 64 from look-up data 48.

[0041] Although the flow chart of Fig. 10 shows a specific order of execution, the order of execution may differ from that which is depicted. For example, the order of execution of two or more blocks may be scrambled relative to the order shown. Also, two or more blocks shown in succession in Fig. 10 may be executed concurrently or with partial concurrence. All such variations are within the scope of the present invention.

Attorney Docket Number
RIDG101

[0042] *SAMPLE COMPRESSED DATA STRUCTURE:* Figs. 11-16 contain block diagrams and tables illustrating a compressed data structure designed for an ARM (Advanced RISC Machine) processor. Fig. 11 illustrates a sample library 50 made up of ten segments 52 containing varying numbers of indexed positions 56. Each segment 52 can be represented by a four bit binary string ranging between 0000 and 1010. The number of positions 56 in each successive segment 52 follows a binary progression increasing by a factor of two. The first segment 52 contains sixteen positions 56 while the tenth segment 52 contains over eight thousand positions 56. Fig. 12 further illustrates the first segment 52 of Fig. 11. In Fig. 12, first segment 52 contains sixteen indexed positions 56. As illustrated, each position can be represented by a four bit binary string ranging from 0000 to 1111. Each successive segment, then, requires one additional bit to identify a particular code string 54 in that segment. In this example, each code string 54 contains thirty-two bits 58.

[0043] As described above with reference to Fig. 8, each look-up string 64 contains an instruction 66 and an index 68. Fig. 13 is a table representing look-up strings 64 designed to cause CPU 28, when executing decompression program 38, to retrieve a code string 54 from library 50 (Figs. 11 and 12) and write that code string 54 without alteration to output memory 42. Instruction 66 includes bits identifying the library segment 52 from which a selected code string 54 is to be retrieved. The index 68, then, contains bits required to identify the indexed position 56 containing the selected code string 54. In this example, the instruction 66 to retrieve a code string 54 from library 50 is a four bit binary string ranging between 0000 and 1001 and the index 68 is a binary string ranging between four bits for the first library segment and thirteen bits for the last library segment. As a note, when read by CPU 28, instruction 66 in the range between 0000 and 1001 also acts as a guide for CPU 28 to retrieve a code string 54 from a particular segment 52 of library 50 rather than from history cache 40 and write that code string 54 without alteration to output memory 42. As can be seen from the table of Fig. 13, each thirty-two bit code string 54 in library 50 can be identified using a look-up string 64 of between eight and seventeen bits. To maximize compression, the sixteen most retrieved code strings 54 are generally found in the first library

segment 52 – the next thirty-two most retrieved in the second segment 52 – the next sixty-four in the third – and so on.

[0044]   Fig. 14 is a table representing look-up strings 64 designed to cause CPU 28, when executing decompression program 38, to retrieve a code string 54 from library 50, alter that code string 54, and then write the altered code string 54 to output memory 42. The following describes the different alterations performed by CPU 28 when reading a look-up string 64 corresponding to each of the four entries in the table of Fig. 14:

1.   An instruction 66 containing the bits 1011 causes CPU 28 to switch a selected bit (0 to 1– or – 1 to 0) of a retrieved code string 54. The first fourteen bits of the index 68 of the particular look-up string 64 identifies a selected code string 54 in any segment 52 and indexed position 56 of library 50. The last five bits of the look-up string 64 identify the particular bit in the code string 54 to be switched.

2.   An instruction 66 containing the bits 1100 causes CPU 28 to perform an arithmetic operation on a retrieved code string 54. The first fourteen bits of the index 68 of the particular look-up string are an identifier pinpointing a selected code string 54 in any segment 52 and indexed position 56 of library 50. The last eight bits are an arithmetic string, in this case, a signed byte to be added to the code string 54. For that byte, the last bit provides the sign for that byte. A one represents a positive number. A zero represents a negative number or vice versa. The remaining seven bits represent the number ranging between –128 and +127 (base ten) to be added to the retrieved code string 54.

3.   An instruction 66 containing the bits 1101 causes CPU 28 to replace a particular byte of a retrieved code string 54. The first fourteen bits of the index 68 of the particular look-up string 64 are an identifier pinpointing a selected code string 54 in any segment 52 and indexed position 56 of library 50. The last eight bits are a replacement – in this case a replacement byte. The particular byte of code string 54 to be replaced is predetermined and set by the instruction 66 of the look-up string 64 and decompression program 38. Potentially, there could be four distinct instructions – each for replacing a different byte of a retrieved code string 54 having 32 bits.

4.   An instruction 66 containing the bits 11100 causes CPU 28 to switch two selected bits (0 to 1 or 1 to 0) of a retrieved code string 54. The first fourteen bits of the index 68 of the particular look-up string 64 identifies a selected code string 54 in any segment 52 and indexed position 56 of library 50. The next five bits of the look-up string 64 identify the first particular

bit in the code string 54 to be switched, while the last five bits of the look-up string 64 identify the second particular bit in the code string 54 to be switched.

As can be seen from the table of Fig. 14, the four groups of look-up strings 64 designed to alter a code string 54 respectively require twenty-three, twenty-six, twenty-six, and twenty-nine bits. Consequently, when representing a code sting of thirty-two bits, use of the look-up strings 64 of Fig. 13 is preferred. However, depending upon the data compressed, use of one or more of the look-up strings 64 represented in Fig. 14 can be provide a significant benefit.

[0045] Figs. 15A and 15B are tables representing look-up strings 64 designed to cause CPU 28, when executing decompression program 38, to retrieve a code string 54 from history cache 40 and then write or duplicate that code string 54 to output memory 42. It is assumed in these examples that each indexed position 60 in history cache 40 can be represented using fourteen bits. The following describes the different alterations performed by CPU 28 when reading a look-up string 64 corresponding to each of the two entries in the table of Fig. 15A:

1. An instruction 66 containing the bits 11101 causes CPU 28 to retrieve a code string 54 from history cache 40 in and indexed position 60 represented by the first fourteen bits of the index 68. Thee last three bits of the index 68 identify the number of times that code string 54 is to be written to output memory 42. The three bits in this example allow the code string 54 to be written up to eight times. A fewer or greater number of bits may be used.

2. An instruction 66 containing the bits 11110 causes CPU 28 to retrieve the most recent entry in history cache 40. The five bits in the index 68 indicate the number of times that code string 54 is to be written to output memory 42. The use of five bits in this example allows the code string 54 to be written up to thirty-two times. A fewer or greater number of bits may be used.

3. An instruction 66 containing the bits 11111 causes CPU 28 to switch a selected bit (0 to 1 or 1 to 0) of a retrieved code string 54. The first fourteen bits of the index 68 of the particular look-up string 64 identifies a selected code string 54 in history cache 40. The last five bits of the look-up string 64 identify the particular bit in the code string 54 to be switched.

[0046] The following describes the different alterations performed by CPU 28 when reading a look-up string 64 corresponding to each of the three entries in the table of Fig. 15B:

1. An instruction 66 containing the bits 11101 causes CPU 28 to retrieve a code string 54 from history cache 40 in and indexed position 60 represented by the first fourteen bits of the index 68. Thee last three bits of the index 68 identify the number of times that code-string is to be written to output memory 42. The three bits in this example allow the code string 54 to be written up to eight times. A fewer or greater number of bits may be used.

2. An instruction 66 containing the bits 11110 causes CPU 28 to switch a selected bit (0 to 1 or 1 to 0) of a retrieved code string 54. The first fourteen bits of the index 68 of the particular look-up string 64 identifies a selected code in history cache 40. The last five bits of the look-up string 64 identify the particular bit in the code string 54 to be switched.

3. An instruction 66 containing the bits 11111 causes CPU 28 to switch two selected bits (0 to 1 – or – 1 to 0) of a retrieved code string 54. The first fourteen bits of the index 68 of the particular look-up string 64 identifies a selected code string 54 in history cache 40. The next five bits of the look-up string 64 identify the first particular bit in the code string 54 to be switched, while the last five bits of the look-up string 64 identify the second particular bit in the code string 54 to be switched.

[0047] Fig. 16 is a table representing look-up strings 64 designed to cause CPU 28, when executing decompression program 38, to write the index 68 of the look-up string 64 to output memory 42.

[0048] The tables of Figs. 11-16 are examples only. The selected operations and sizes of library 50 and look-up strings 64 were designed for an ARM processor. In this case, library 50 contains approximately 16,000 code strings 54 each containing thirty-two bits. With a library of this size, as CPU 28 executes decompression program 38, the most accessed code-strings 54 in library 50, if not the entire library 50, naturally migrate to processor cache 46. The same is true for the most repeated instructions of decompression program 38. Consequently, CPU 28 is able to inflate compressed data structure 36 accessing volatile and non-volatile memory as few times as possible. For portable computing systems utilizing an ARM processor, this feature increases the speed at which a program or other data is inflated resulting in better performance and an increased battery life.

[0049] However, the specifics for any given library 50 and set of look-up strings 64 is dependent upon many factors including the type of CPU 28 involved, the compiler used when writing the compressed program, and the specifics of the

program itself. Code strings 54 need not contain 32 bits. Look-up strings 64 can contain any number of bits. Library 50 can be broken into any number of segments 52 with each segment 52 containing any number of positions 56. History cache 40 can contain any number of indexed positions 60. The types operations performed on code strings 54 are not limited to those described above with reference to Figs. 13-16. Depending upon the number of bits 58 in each code string 54 and the number of entries in library 50 and history cache 40, it may be desirable to provide an operation for switching three selected bits rather than one or two. It may be desirable to replace two or more bytes of a given code string 54 rather than just one. Moreover, it may be beneficial in some cases to perform an arithmetic operation in which a number greater than 128 is added to or subtracted from a given code string 54. The particular bits in the instruction 66 of a look-up string 64 that identify any given operation are unimportant. The most repeated operations, however, should be represented with the fewest bits.

[0050]   The present invention has been shown and described with reference to the foregoing exemplary embodiments. It is to be understood, however, that other forms, details, and embodiments may be made without departing from the spirit and scope of the invention which is defined in the following claims.